Name: Andrew Hayes
Student ID: 21321503
Programme: 4BCT

CT4101

2024–11–24

Assignment 2: Regression Using Scikit-Learn

1 Description of Algorithms

1.1 Algorithm 1: Random Forest

Random forest is a supervised machine learning algorithm used for classification and regression tasks. It builds upon the **decision tree** algorithm by constructing a collection of decision trees, where each tree is trained independently on random subsets of the data. These trees are then aggregated to make the final prediction. An implementation of this algorithm for regression is provided in scikit-learn as `sklearn.ensemble.RandomForestRegressor`[6].

To overcome the problems of decision trees being unstable and causing overfitting, the random forest algorithm combines many randomly-generated decision trees into a single forest to improve accuracy, reduce overfitting, & reduce variance. Each tree is built independently on a different random subset of the training data and features, which helps reduce variance and overfitting. The final prediction is obtained by averaging the predictions of each individual tree (for regression tasks) or by majority voting (for classification tasks). The accuracy of the model in random forests is improved through **bagging**, a technique that averages predictions over a large number of independently trained trees to reduce variance and prevent overfitting.

In `RandomForestRegressor`, the trees are constructed as follows:

1. The algorithm generates multiple bootstrap samples by randomly selecting subsets of the training data with replacement. These subsets are used to train individual decision trees. Selecting the samples with replacement allows the generated trees to overlap, which helps to reduce variance and overfitting.
2. Each node is split to partition the data, decided by a random subset of the features. The split that causes the greatest reduction in error according to the mean squared error algorithm is chosen. This process is repeated for every node starting at the root node until there are no splits left to be made.
3. The final prediction is made by averaging the predictions of all the individual trees in the forest (for regression tasks).

I chose the random forest regressor because it is robust to overfitting, handles high-dimensional datasets well, and offers strong performance without requiring extensive tuning of hyperparameters. It is also an interesting comparison to the gradient boosting regressor, as both are ensemble methods that leverage multiple decision trees but with different strategies for combining these trees.

1.1.1 Hyperparameter 1: `n_estimators`

The hyperparameter `n_estimators` is an ***`int`*** with a default value of 100, which controls the number of decision trees added to the model[6]. Increasing the number of estimators generally improves model accuracy and stability, but it may lead to overfitting if too many trees are added. Thus, there is a trade-off between performance and computational cost, as more estimators require more training time and resources.

1.1.2 Hyperparameter 2: `max_depth`

The hyperparameter `max_depth` is an ***`int`*** with a default value of **`None`**, controlling the maximum depth of each tree in the ensemble[4]. By limiting the depth, the algorithm helps prevent overfitting, as shallower trees capture more general patterns in the data. High `max_depth` values allow trees to capture complex patterns, but can lead to overfitting and reduced generalisation on unseen data. In contrast, lower `max_depth` values create simpler trees that focus on the most important features and relationships, which can enhance generalisation but may result in underfitting if the model fails to capture necessary data complexity.

1.2 Algorithm 2: Gradient Boosting

Gradient boosting is a supervised machine learning algorithm that can be used for both classification & regression tasks. It builds upon the **decision tree** algorithm by combining multiple decision trees sequentially, with each new tree trained to correct the errors of the previous ones. This sequential training process is distinct from that of **random forests**, which aggregates many trees independently trained on random subsets of the data. An implementation of this algorithm for regression is provided in scikit-learn as `sklearn.ensemble.GradientBoostingRegressor`[4]. While gradient boosting can also be used for classification, I will only refer to its use as a regression algorithm in this assignment, as classification is not relevant to the specific task at hand.

While both random forest and gradient boosting use multiple decision trees, they differ fundamentally in how they build and combine these trees:

- In **Random Forest**, trees are built independently on random subsets of data and features, which helps reduce variance and improves generalisation. The final prediction is obtained by averaging the outputs of each tree (for regression tasks) or by majority voting (for classification tasks).
- In **Gradient Boosting**, trees are built sequentially, with each new tree trained to correct the errors of the previous ones. This iterative process focuses on reducing bias, as each tree addresses the residual errors from prior iterations. The final prediction is obtained by summing the predictions of each tree, where later trees contribute to refining the overall model. This general process of creating new models to sequentially correct the errors of previous iterations is called **boosting**: these iterative models are called *weak learners* (in this context, the decision trees are the weak learners) and are combined to form what is (hopefully) a *strong learner*. These weak learners are typically only somewhat more accurate than random guessing, and are often very shallow trees, but when combined can produce very accurate results.

The accuracy of the weak learners is assessed with a **loss function** which quantifies the error in the model. While `GradientBoostingRegressor` provides different options for the loss function used, the default is `loss = 'squared_error'` or Mean Squared Error, which is a domain-specific measure of error given by:

$$\text{MSE} = \frac{\sum_{i=1}^{n}(t_i - \mathbb{M}(d_i))^2}{n}$$

where $\mathbb{M}(d_1) \ldots \mathbb{M}(d_n)$ is a set of n values predicted by the model and $t_1 \ldots t_n$ is a set of labels.[7]

In `GradientBoostingRegressor`, the trees are generated as follows:

1. The algorithm initializes the model with a constant value, which serves as the baseline prediction for all instances.
2. At each iteration, the algorithm computes the negative gradient of the loss function with respect to the current model's predictions. This gradient indicates the direction in which to adjust the predictions to minimize the loss.
3. A decision tree is then fitted to the negative gradients (i.e., the errors) from the previous iteration, where each tree is constrained in complexity (often by limiting the depth) to avoid overfitting.
4. The predictions from this new tree are scaled by a learning rate and added to the ensemble's predictions to update the model.

I chose the gradient boosting regressor because it is effective for complex and non-linear datasets, provides robust performance on diverse types of data, and offers flexibility with numerous hyperparameters for tuning. It is also an interesting comparison to the random forest regressor, as they both are ensemble methods that use multiple decision trees.

1.2.1 Hyperparameter 1: `n_estimators`

The hyperparameter `n_estimators` is an ***`int`*** with a default value of 100, which controls the number of boosting stages (i.e., decision trees) added to the model[4]. Increasing the number of estimators generally improves model accuracy and stability, but it may lead to overfitting if too many trees are added. Thus, there is a trade-off between performance and computational cost, as more estimators require more training time and resources.

1.2.2 Hyperparameter 2: `max_depth`

The hyperparameter `max_depth` is an ***`int`*** with a default value of 3, controlling the maximum depth of each tree in the ensemble[4]. By limiting the depth, the algorithm helps prevent overfitting, as shallower trees capture more general patterns in the data. High `max_depth` values allow trees to capture complex patterns, but can lead to overfitting and reduced generalisation on unseen data. In contrast, lower `max_depth` values create simpler trees that focus on the most important features and relationships, which can enhance generalisation but may result in underfitting if the model fails to capture necessary data complexity.

2 Model Training & Evaluation

For both algorithms, I first trained a model with the default hyperparameters using 10-fold cross-variation to get a baseline to which I could compare my results. To achieve this 10-fold cross-validation, I used the `sklearn.model_selection.KFold` and the `cross_validate` function which I provided with a seed of `random_state=42` to get consistent testing results. I then tuned my chosen hyperparameters using the `GridSearchCV` class. I performed two grid searches for each algorithm, one with each of my chosen measures of error.

2.1 Measures of Error

2.1.1 Domain-Specific Measure of Error

I chose **mean squared error** as my domain-specific measure of error, as it is simple & intuitive to understand, penalises larger errors more strongly than weaker errors, and because it is the default loss function used in `GradientBoosingRegressor`. Since I did not choose the loss function of `GradientBoosingRegressor` as a hyperparameter to tune for this assignment, the `GradientBoosingRegressor` will attempt to optimise the MSE of model by default and thus I feel that it would be most appropriate to also use this metric in my own analysis; it doesn't seem effective to tune the model to optimise for one measure of error, and then actually assess the model by using an entirely different measure. Similarly, `RandomForestRegressor` uses MSE to determine the optimal split at a node in the decision trees it creates.

The equation by which MSE is defined can be found above in **§1.2 Algorithm 2: Gradient Boosting**.

2.1.2 Domain-Independent Measure of Error

I chose the R^2 coefficient as my domain-independent measure of error primarily due to it being so intuitive & simple to understand. It works by imagining there exists a model that always predicts the average values from the test set, and compares the model in question to this imaginary model. The R^2 coefficient is given by:

$$R^2 = \frac{\frac{1}{2}\sum_{i=1}^{n}\left(t_i - \mathbb{M}\left(d_i\right)\right)^2}{\frac{1}{2}\sum_{i=1}^{n}\left(t_i - \bar{t}\right)^2} = \frac{\text{sum of squared errors}}{\text{total sum of squares}}$$

where $\bar{t}$ is the average value of the target variable.[7]

2.2 Algorithm 1: Random Forest

```
Average Training MSE: -119.6467
Average Testing  MSE: -814.3854

Average Training  R²: 0.9856
Average Testing   R²: 0.8959
```

Listing 1: Average training & testing MSE & R^2 error scores with default hyperparameters

As can be seen from the above output of my Python program, the average MSE score increased significantly when testing the model compared to how it performed on the test data. While a reduction in accuracy is to be expected, this could indicate that there was some slight overfitting to the training data. The reason that the MSE scores are negative as they are being used as a scoring metric, and scikit-learn negates the values so that they are consistent with other scoring metrics. Overall, the results with the default hyperparameters were very good and produced very low error margins.

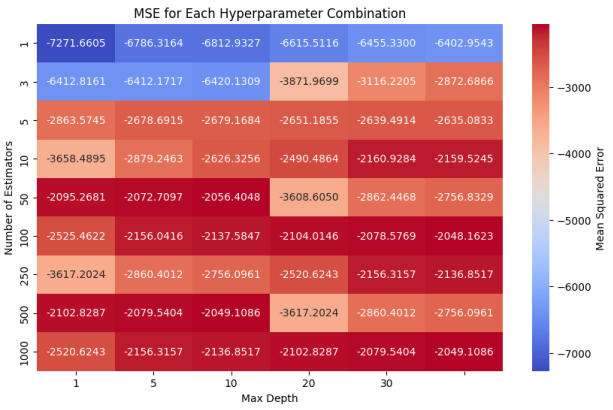

Figure 1: Heatmap of MSE error scores

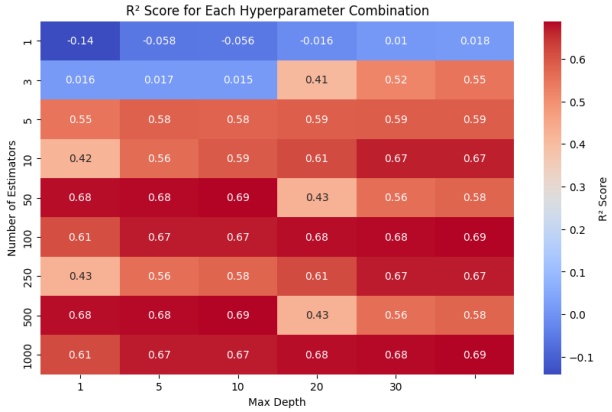

Figure 2: Heatmap of R^2 error scores

As can be seen from the above heatmaps, the best error scores I managed to achieve with hyperparameter tuning were a MSE of approximately -2049 and a corresponding R^2 of 0.69, significantly lower than the scores achieved with the default hyperparameter values, achieved with a `max_depth` of **`None`** and a `n_estimators` of 100. In this sense, the hyperparameter tuning can be considered a failure as it did not succeed in increasing the accuracy beyond the default values. One possible reason for this is an insufficiently fine-grain search: the optimal results appear in the 50-500 range for the `n_estimators` hyperparameter and a search that focused more on this range could potentially have yielded better results. However, even the 9×6 level of granularity in searching seen here took around 15-20 minutes to run on my hardware, making finer-grain searching difficult and time-consuming.

2.3 Algorithm 2: Gradient Boosting

```
Average Training MSE: -287.2685
Average Testing  MSE: -783.5938

Average Training  R²: 0.9653
Average Testing   R²: 0.9020
```

Listing 2: Average training & testing MSE & R^2 with default hyperparameters

Gradient boosting with the default hyperparameters performed slightly worse than random forest, but had less increase in error in the testing data, lending some credence to the idea that the random forest algorithm may have been slightly overfitting the data. However, the error scores were nonetheless very good.

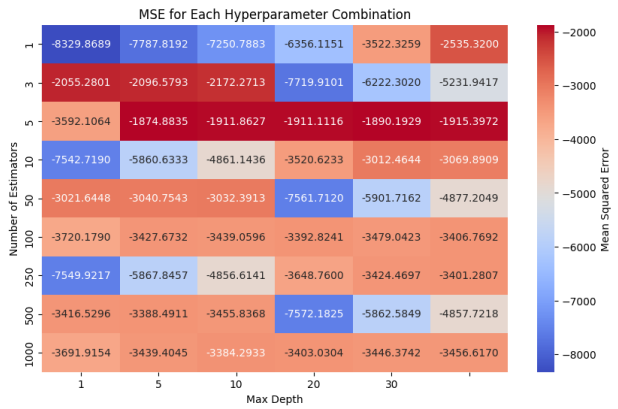

Figure 3: Heatmap of MSE error scores

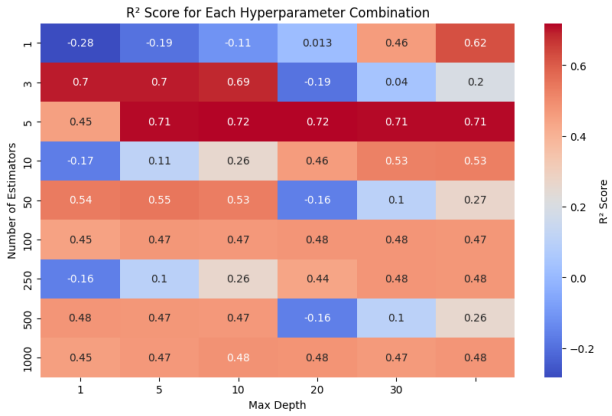

Figure 4: Heatmap of R^2 error scores

As can be seen in the above heatmaps, I again failed to improve on the accuracy achieved by the default hyperparameter values. Part of this is almost certainly due to the range of values searched through being poor: to maximise comparability with the random forest algorithm I used the same search range, but a glance at the heatmaps shows that it performed very poorly with `n_estimator` values past 5. A better search would likely have focused on a smaller range of `n_estimator` values. The best error scores obtained were a MSE of -1911.116 and a R^2 of 0.72, beating the random forest hyperparameter tuning, achieved with a `max_depth` of 20 and a `n_estimators` of 5. The much lower `n_estimators` of 5 here compared to random forest's 100 serves to illustrate a key difference between the two algorithms: random forest performs well by aggregating many trees covering random, overlapping subsets of the data, so a higher number of trees can increase stability and reduce variance. The gradient boosting algorithm, on the other hand, makes incremental improvements to the estimators, each one trying to correct the errors of the previous one, allowing it to attain higher accuracy with fewer, more precise trees.

3 Conclusion

The key findings are as follows:

- For both algorithms, the default hyperparameters out-performed my attempt at hyperparameter tuning. Thus, I can only recommend the default hyperparameter values for both algorithms.
- With the default hyperparameters, `RandomForestRegressor` out-performed `GradientBoostingRegressor`.
- However, the best hyperparameter tuning results obtained by `GradientBoostingRegressor` out-performed the best tuning results obtained by `RandomForestRegressor`.
- An exhaustive grid search of 10-fold cross-validated models is highly computationally intensive, and thus very time-consuming. Therefore, choosing a small but appropriate range of potential hyperparameter values is key to success if hardware or computation time is limited.

References

[1] Leo Breiman. "Random Forests". In: *Machine Learning* 45 (2001).

[2] scikit-learn Documentation. *`cross_validate`*. URL: `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_validate.html` Accessed on: 2024-11-24.

[3] scikit-learn Documentation. *Ensembles: Gradient boosting, random forests, bagging, voting, stacking*. URL: `https://scikit-learn.org/stable/modules/ensemble.html` Accessed on: 2024-11-24.

[4] scikit-learn Documentation. *`GradientBoostingRegressor` API Reference*. URL: `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html` Accessed on: 2024-11-24.

[5] scikit-learn Documentation. *Model Evaluation*. URL: `https://scikit-learn.org/stable/modules/model_evaluation.html` Accessed on: 2024-11-24.

[6] scikit-learn Documentation. *`RandomForestRegressor` API Reference*. URL: `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html` Accessed on: 2024-11-24.

[7] Frank Glavin. "Machine Learning: Regression". Lecture slides, University of Galway. 2024.

[8] IBM. *What is random forest?* URL: `https://www.ibm.com/topics/random-forest` Accessed on: 2024-10-06.

[9] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.